Random Forests with MapReduce

This report describes an implementation of random forests with MapReduce using ensemble learning

Jerry Gu

University of Waterloo, jy4gu@uwaterloo.ca

1. INTRODUCTION

Machine learning algorithms involve using training data to train a model that can make predictions on test data. However, the amount of memory required for training/testing these models usually scales with the amount of data; therefore, the standard implementations of such algorithms involving a single computer may not be scalable. If the data is large, it is necessary to involve big data algorithms.

When the data is large, a natural solution would be to use ensemble learning: split the training data into N pieces, apply the machine learning algorithm on each piece to obtain N predictions (or distributions of the predictions) which we combine and choose the class that is predicted the most. Compared to just using one split to train the model, ensemble learning can reduce the variance of the estimates (on average, the variance is proportional to $N^{-0.5}$).

The machine learning algorithm of interest in this paper is random forests. It is one of the popular machine learning algorithms, as it is simple yet accurate. It involves using several decision trees, with each one trained on a sample obtained through bootstrap aggregation (bagging) on the training data, and the prediction is the class with the most votes.

1.1. Goal

The goal of this project is to implement a scalable random forest algorithm with MapReduce, by using ensemble learning, and then evaluating it using cross-validation. The learning objective is about scalable machine learning algorithms, using this implementation as an example.

2. METHODOLOGY - TRAINING

For the training class, TrainRandomForest, boilerplate was copied from WordCount, a Java file from Bespin [1]. The following subsections will explain each part of the implementation.

2.1. Parsing the data

Within each mapper, the setup just initializes the dataset (represented by ArrayListWritable<DoubleArrayWritable>) to store the data, and initalizes variables that describe how to parse the data (e.x. which columns to exclude, and which column is the response).

The map function parses each line (of the piece of input file given to its mapper) to a DoubleArrayWritable of length d+1 (the first d items represents the d features, and the last item represents the response) and added to the dataset. The map function doesn't output anything; only the cleanup function outputs things.

2.2. Building the trees

In the cleanup function, the decision trees are built and outputted with the key being a random number (so if there's multiple reducers, the output doesn't all go to only one of them) and the value being the hashmap (as a string) that represents the tree.

Each node in decision trees can either represent a split or a leaf. A split has 2 children and its value is a tuple (i,j) representing the question "is the i-th feature of the observation less than j?" If yes, go to the left child; otherwise, go to the right. Meanwhile a leaf has 0 children and its value is a tuple (-1,j), where j represents which class to predict. Therefore, the trees are binary trees, which can be represented by a list, where the root is at index 1, and if the node at index i has children, then the indices of the children are 2*i for the left and 2*i+1 for the right. However, the size of the list scales exponentially in the height of the tree, and the list is typically more sparse as you go down the tree, so using a hashmap would be more appropriate than a list, where the keys are the indices.

For each tree, a subset of the features are selected by using simple random sampling without replacement, and a bootstrap sample (i.e. a sample created using simple random sampling with replacement) from the dataset created in the mapper. Then, the tree is built by calling a recursive helper function "buildTree", where one of the parameters represent a set of observations used to build a node (let this be X); the first node built is the root, where X is the whole bootstrap sample. To prevent overfitting which causes high variance, the user may pass arguments (in the command line) that set the maximum depth of the tree and minimum number of observations per leaf.

Entropy is used to measure the loss (i.e. expected error) from each split, which can be calculated by: $-\Sigma_i p_i \log(p_i)$, where p_i is the sample proportion of the i-th class in X. Information gain from a split is then how much the entropy decreases by splitting a node.

For each node being built in the tree, if it is at the maximum depth, its entropy is 0, or no valid split can result in information gain, then it is made a leaf node, and its value is whichever class appears the most in X. Otherwise, it is made a splitting node whose value (i,j) is such that the information gain is maximized by splitting on the i-th feature at j, while X1 and X2's sizes are both atleast the minimum number of observations per leaf, where X1 is the observations in X such that the values of their i-th feature is less than j and X2 is the other observations. Then, buildTree is called twice; the first one passes X1 to build the left child, and the second one passes X2 to build the right child.

2.3. Output

The reducers just output whatever the mappers outputted. Therefore, the files in the output folder contain a total of M*N lines, where M is the number of trees per mapper and N is the number of mappers. Each line has a key which is a random number (not used anymore) and a string value representing the decision tree.

3. METHODOLOGY - TESTING

For the testing class, ApplyRandomForest, boilerplate was copied from WordCount, a Java file from Bespin [1]. The following subsections will explain each part of the implementation.

3.1. Parsing the data

Within each mapper, the setup initalizes variables that describe how to parse the data (e.x. which columns to exclude, which column is the response, and which column represents the IDs of the observations). It also loads the output of TrainRandomForest as side data, which is parsed to 2 lists of M*N hashmaps; one to represent the feature to split on or if it's a leaf node (-1 in that case), and the other to represent where to split at or which class to predict. The pair of hashmaps at the i-th value of these lists represent the i-th tree; let these be featuretrees[i] and splittrees[i], respectively.

The map function parses each line (of the piece of input file given to its mapper) to a double array representing the d features (let this be x), and an int representing the response (let this be y) of the corresponding observation. It then makes a prediction on that observation (described below).

3.2. Predicting using the trees

When making a prediction on an observation, each of the M*N trees are traversed in the following way: for the i-th tree at a node with key k, if the node is a leaf (i.e. featuretrees[i]=-1), then that tree predicts the class splittrees[i].get(k); otherwise, determine if feature featuretrees[i].get(k) of the observation is less than splittrees[i].get(k); if yes, then go to the node with key k*2 (left child), otherwise go to the node with key k*2+1 (right child).

Using the resulting count of the number of predictions for each class, we can then predict the observation by choosing which one has the highest count. This is outputted as the value, with the corresponding key being the ID of the observation.

Furthermore, these counts can be used to update class variables: cross entropy, total tests (i.e. test set size), and correct tests. Cross entropy for one prediction is calculated by $-\Sigma_i y_i \log(p_i)$, where p_i is the sample proportion of trees that predicted the i-th class and y_i is the indicator for the actual class. The total cross entropy is simply this sum for all predictions. Thus, we increase cross entropy by that of the current prediction, the total tests by 1, and the correct tests by 1 iff the prediction was correct.

In the cleanup, the cross entropy, total tests, and correct tests are outputted as values with corresponding keys -1, -2, and -3, respectively. It is assumed the ID's are non-negative. Using a partitioner, these are sent to a separate reducer.

3.3. Output

The reducers just output whatever the mappers outputted but with the value (i.e. the prediction, which was mapped to a number before) mapped back to the corresponding label name. Except when the key is <0, in which case nothing is outputted: if the key is -1, -2, and -3, then it increments cross entropy, total tests, and correct tests by the values, respectively.

In the cleanup, the cross entropy and accuracy is printed for the reducer that was sent the keys -1, -2, -3, where the accuracy is the correct tests divided by total tests.

Therefore, the file will have each of the test observations' ID's along with their predicted class.

4. EVALUATION

The setup is similar to that of the cs651 assignments; i.e., in the folder there should be the pom.xml file, the TrainRandomForest.java and ApplyRandomForest.java files, then do "`mvn clean package`" to build.

4.1. On Iris dataset

The popular Iris dataset [2] will be used to evaluate the results. This dataset has 6 columns: Id, SepalLengthCm, SepalWidthCm, PetalLengthCm, PetalWidthCm, Species. We let the 1st column be the ID, the 6th column be the response, and the rest be the features.

As an example, the 1st observation of every 10 (i.e. those with ID's 1, 11, 21, 31, 41, etc.) were put in the test set (named "data/iristest"), and the rest were put in the training set (named "data/iristrain"). Furthermore, suppose we want 3 reducers, 20 trees trained on 2 features each, with maximum depth of 10 and minimum 3 observations per leaf, and initial seed 1 (set the seed for reproducibility), then we do the following (for ease of reading, omitted logs except for cross entropy and accuracy, and omitted the side data outputs except the first 2 hashmaps):

```
  $ hadoop jar target/assignments-1.0.jar
ca.uwaterloo.cs451.project.TrainRandomForest \
```

```
>    -input data/iristrain -output sidedata -reducers 3 -exclude 0 -yidx 5
-labels Iris-setosa,Iris-versicolor,Iris-virginica -seed 1 -features 2 -numtrees
20 -minnodes 3 -maxdepth 10
…
$ cat sidedata/part-r-00000
-1535946032     {1=2,2.5999999999999996, 2=-1,0.0, 3=2,4.75, 6=-1,1.0, 7=2,5.15,
14=0,6.5, 15=-1,2.0, 112=-1,2.0, 113=-1,2.0, 56=0,5.9, 57=-1,1.0, 28=0,6.25,
29=-1,1.0}
-744030269{1=2,2.45, 2=-1,0.0, 3=3,1.75, 6=2,4.95, 7=-1,2.0, 12=-1,1.0,
13=-1,1.0}
…
$ cat sidedata/part-r-00001
…
$ cat sidedata/part-r-00002
…
$ hadoop jar target/assignments-1.0.jar
ca.uwaterloo.cs451.project.ApplyRandomForest \
>    -input data/iristest -sidedata sidedata -output out -reducers 3 -ididx 0
-yidx 5 -labels Iris-setosa,Iris-versicolor,Iris-virginica
…
2022-12-14 02:55:19,155 INFO project.ApplyRandomForest: cross-entropy:
2.7784116425825744
2022-12-14 02:55:19,155 INFO project.ApplyRandomForest: accuracy:
0.9333333333333333
…
$ cat out/part-r-00000
21   Iris-setosa
51   Iris-versicolor
81   Iris-versicolor
111  Iris-virginica
141  Iris-virginica
$ cat out/part-r-00001
1    Iris-setosa
31   Iris-setosa
61   Iris-versicolor
91   Iris-versicolor
121  Iris-virginica
$ cat out/part-r-00002
11   Iris-setosa
41   Iris-setosa
71   Iris-virginica
101  Iris-virginica
131  Iris-virginica
```

From the outputs, we see that 14 out of 15 test observations were predicted correctly (the only wrong prediction is ID 71; it predicted Iris-virginica, but it's actually Iris-versicolor). We perform 10-fold cross-validation by repeating the above procedure 9 more times, except with the i-th observation of every 10 (e.x. for i=2, those with ID's 2, 12, 22, etc.) being put in the test set (and the rest in the training set) and the seed being set to i, for i=2, 3, … , 10. These results are summarized in the table below. Each of them predict atleast 13 of 15 test observations correctly. Considering the 3 classes are evenly balanced in the dataset (as there are exactly 50 of each), accuracy is a reasonable metric for prediction here. Therefore, we see that the algorithm is quite accurate.

i	Cross-entropy	Accuracy
1	2.77841164258257	0.9333333333333333
2	1.0432225248429137	1.0
3	2.0537737949312582	0.9333333333333333
4	3.1975220574620993	0.8666666666666667
5	2.343812579749346	0.9333333333333333
6	0.5469224712722589	1.0
7	3.6375439606167803	0.9333333333333333
8	2.607924543435623	0.9333333333333333
9	2.1418348135110237	0.9333333333333333
10	2.29615697580462	0.933333333333333

Table 1: cross-validation results

5. RELATED WORK

Boilerplate in the implementations were copied from Bespin [1]. Iris dataset [2] was used for evaluation.

REFERENCES

1. cs451. 2017. Bespin is a library that contains reference implementations of "big data" algorithms in MapReduce and Spark. Retrieved December 9, 2022 from https://git.uwaterloo.ca/cs451/bespin/-/blob/master/src/main/java/io/bespin/java/mapreduce/wordcount/WordCount.java
2. UCI Machine Learning. 2016. The Iris dataset was used in R.A. Fisher's classic 1936 paper, The Use of Multiple Measurements in Taxonomic Problems. Retrieved December 10, 2022 from https://www.kaggle.com/datasets/uciml/iris